Developing efficient heuristic approaches to cluster editing, inspired by other clustering problems

Angelos Zoumis

Published: 27 June 2021
Delft University of Technology
Supervisor: Dr Emir Demirović

Abstract

Cluster editing attempts to find the minimum number of edge additions and removals on an undirected graph, that will transform the graph to one consisting of only disconnected cliques. In this paper, we propose three heuristic approaches to this problem, based on algorithms used to solve different clustering problems. The algorithms were based on agglomerative, divisive and k-means clustering algorithms. Experimental results show that all three algorithms are able to find results close to the minimum number of edits, but in particular, the k-means algorithm has a lower time complexity compared to the other two algorithms, while producing on average, the lowest number of edits.

Keywords: Agglomerative hierarchical clustering, closed neighborhood, Cluster editing, Clustering, Divisive hierarchical clustering, Dijkstra's algorithm, distance metrics, K-means

1 Introduction

Clustering is a class of problems that attempts to group together similar data points into clusters. One such problem is cluster editing. In detail, it is the problem of, given an undirected graph, find what is the minimum number of edge edits (edge additions and removals), that should be performed, in order to transform the graph into a cluster graph, meaning a graph consisting of only disconnected cliques. An example of the cluster editing problem is shown in figure 1.

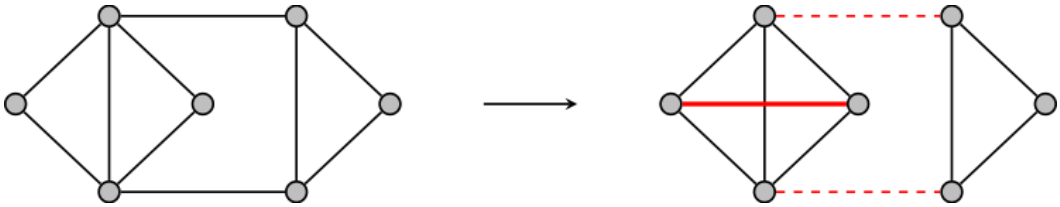

Figure 1: Transforming a graph into a cluster graph. The red and striped lines represent the edge additions and removals respectively. For this graph, 2 removals and 1 addition need to be performed in order to transform the graph into a cluster graph. Hence, we need 3 edits to perform the cluster editing problem ("PACE 2021", 2021)

This problem has many different applications, like studying proteins in biology (Röttger et al., 2012; Wittkop, Rahmann, Röttger, Böcker, & Baumbach, 2011). Hence, being able to solve this problem fast for any size of graph can be of great benefit to society. Unfortunately, this problem has been

proven to be NP-Hard, resulting in exact solvers being unable to find an answer on larger graphs in a reasonable time (Rokach & Maimon, 2005).

Due to the high complexity of finding the exact solution, a heuristic algorithm could be beneficial. A heuristic algorithm would not guarantee that the final result would produce the minimum number of edits, however, an algorithm with lower time complexity would produce a result much faster, as the size of the graph increases. This would make finding a cluster editing solution for larger graphs feasible, improving the usability of such an algorithm.

To create such an algorithm, one solution would be to look at similar problems, and adjust the algorithm, in order to fit the requirements of the cluster editing problem. A solution based on previous algorithms would also mean that developing the code can be faster, as the algorithm is based on already existing work. Other clustering problems, mainly clustering on a Euclidean space could be adjusted to fit the goal of cluster editing. Due to the importance of this problem in unsupervised machined learning, a lot of different approaches exist for this problem (Rokach & Maimon, 2005).

The goal of this research is to determine what are the main differences that prevent us from using clustering algorithms for other problems to solve the cluster editing problem. Next, the aim is to attempt to address all the mentioned issues, and develop effective heuristic algorithms, inspired by existing clustering algorithms. Last through experimentation, the research will attempt to discover which developed algorithm is the best for solving the problem, in terms of time and proximity to the exact answer. The final result of this research will be an algorithm that is able to efficiently solve the cluster editing problem in a heuristic approach.

The report is structured as such: Section 2 discusses the methodology that was used for the research, and how the algorithms were developed and evaluated. This includes how the performance was measured, as well as what comparisons were made. Next, in section 3, the algorithms that are the basis of the final code are explained. Furthermore, this section also details the issues that should be addressed in order for the algorithms to fit the needs of this problem. Next section 4 discusses what modifications were made to the algorithms, in order for them to be able to solve the cluster editing problem. Section 5 details the setup of the experiment. In particular, this section expands on section 2, and focuses on the more technical parts of the experiment, like the platform that was used to run the experiment. Continuing, section 6 contains and analyzes the results of the experiment. Section 7 reflect on the ethical aspects of the research, by providing some examples where this algorithm could be useful, and addressing the reproducibility of the research. Finally, section 8 contains the concluding remarks, as well as future recommendations on what more can be improved on this topic.

2 Methodology

The goal of the research is to see if a heuristic algorithm can be developed for the cluster editing problem, inspired by clustering algorithms. In this section, it is discussed how the algorithms are compared, and what the experiments attempt measure.

First, a theoretical analysis of the algorithms should occur. The goal of this is to determine their complexity, how they compare to other algorithms, and also discuss what advantages and disadvantages we expect to see for each algorithm. More importantly, the algorithms go through an experimental testing phase. By performing several experiments, this showcases if the theoretical analysis done reflects the real-world performance.

The independent variables for the research are the following: First, the algorithms developed for this research. Furthermore, variables inside the algorithms, like distance metrics, are also adjusted, in order to observe how they affect the final performance. Next 100 graphs, provided by the "exact path" of PACE 2021 (2021) are used. The "exact path" contains smaller graphs, which can be solved by an exact solver in a reasonable time. The reason for using the smaller graphs of the PACE challenge, is that the exact solution is known, and the time it takes to run the algorithms for each test is smaller. This allows us to run the tests multiple times, in case something goes wrong during the experiment, or if changes in the code were made.

The dependent variables for the experimental work are the number of edits and time taken to find the final solution. These two variables allow us to compare how well the algorithms compare to the optimal solution, and between each other, while also allowing us to see which algorithm is faster, and what trend they follow as the size of the graph increases.

Lastly, the controlled variables are the language, the interpreter, and the computer. The algorithms are written in python, with the addition of NumPy. Python is preferred, as it allows us to easily edit code and translate the algorithms from pseudocode to python, and back. As important, all the experiments are run on the same computer and interpreter, in order to avoid any unfair advantages between tests.

3 Algorithms

The first step in developing the final code is to determine which algorithm will be used as a basis. In this section, the k-means clustering, and hierarchical clustering methods are investigated. The goal is to understand how these methods work, and what issues should be addressed, in order for these algorithms to work on the cluster editing problem.

3.1 K-means clustering

K-means clustering is a popular clustering algorithm. In the work of Hamerly & Elkan (2002), a lot of ways to perform k-means algorithm are detailed. The most common implementation of the algorithm is based on Lloyd's algorithm (Lloyd, 1982), which can be described as such in pseudocode:

1. Generate the initial set of k means.
2. Assign each point to the cluster with which the squared Euclidean distance is the smallest.
3. Using the new points in the clusters, recalculate the new means for each cluster, and repeat step 2.

This algorithm will converge when after each iteration, step 2 produces the same cluster. The algorithm does not guarantee the optimal solution. The time complexity of the algorithm is $\mathcal{O}(kidn)$, with k being the number of clusters, i being the number of iterations, d being the dimensions of the Euclidian space, and n being the number of points.

In order for this algorithm to work, the following questions should be addressed: First, a k value needs to be created, which should reflect the number of clusters we expect the final solution to have. Second, a way to generate the initial set of k means needs to be discovered. Continuing, since the cluster editing problem is performed in a graph, and not a Euclidean space, a different measurement of distance should be used. Last, a way to update the k-means in step 3 is needed.

3.2 Hierarchical clustering

Another approach to clustering is through hierarchical clustering. In general, there are two approaches to hierarchical clustering (Rokach & Maimon, 2005).

First, an agglomerative method could be used. This approach starts with each point in its own cluster, and continuously merges each cluster, until there is one cluster.

Second, the divisive approach works in the opposite way. It starts with a single cluster, and keeps dividing until each point is its own cluster.

The hierarchical approach should be easier to translate into the cluster editing problem, as fewer issues need to be addressed. Mainly, it solves the issue of having to predetermine the number of clusters beforehand, and determining how to re-calculate the means of the clusters. However, for this approach to work, it needs to be discussed what metrics should be used to divide or merge the clusters.

4 Modifying the algorithms

The goal of this section is to address the issues discussed in section 3 of the report. First, the issue of measuring the distance is addressed, and next, each algorithm is modified to solve the cluster editing problem.

4.1 Distance metrics

The goal of a distance metric in any clustering problem is to reflect how likely it is for certain points to belong in the same cluster. Therefore, when developing said metrics, the aim should be for vertices that are in the same cluster in the optimal solution to have a smaller distance than the ones that are in different clusters.

4.1.1 Dijkstra

The naive approach is to measure the shortest edge path between the two vertices in order to determine the distance between two vertices. We can find the solution using Dijkstra's algorithm (1959).

This algorithm could be used to determine whether two or more vertices belong in the same cluster. It makes sense that vertices that have a larger distance are more likely to be in different clusters. In fact, from the research of Bastos et al. (2014) we know that for any two vertices with distance greater than or equal to 3, they should be in different clusters. More formally, we get the following proposition:

Proposition 1 *Let G be a graph, with i and j being vectors of G. If the distance(i, j) ≥ 3, then, in all optimal solution of the cluster editing problem for G, vertex i and j are parts of different clusters.*

Therefore, using Dijkstra's algorithm, we can determine that vertices with distance greater than 3 should be split up. However, this algorithm has many flaws that make it not the best approach for this problem. First, for vertices with distance less than 3, this algorithm cannot find which vertices should belong in the same cluster. Furthermore, the time complexity for Dijkstra is $\mathcal{O}(|E| + |V|log|V|)$, where E is the number of edges, and V is the number of vertices. This time complexity could be improved, by using a different distance metric.

From proposition 1, a new observation can be made. In particular, if two vertices i and j that have a distance greater than 3, they are guaranteed to share no vertices directly connected to both of them through an edge, since that would imply that there is a path with distance of 2, and are not directly connected themselves, since that would result in a distance of 1. The set of all vertices that are directly connected to a vertex i, plus i itself are referred as the closed neighborhood, and is denoted as $N_G[i]$. Therefore, we can restate proposition 1 as such:

Proposition 1 restated *Let G be a graph, with i and j being vectors of G. If $N_G[i] \cap N_G[j] = \emptyset$, then, in all optimal solution of the cluster editing problem for G, vertex i and j are parts of different clusters.*

From this, we can conclude that in order to determine if two vertices should belong in the same or different clusters, we could only look at their closed neighborhoods.

4.1.2 Closed neighborhood similarity

The next metric uses the similarity of the closed neighborhood of the vertices to determine the distance between vertices. In particular the following formula is used:

$$d_G(i,j) = 1 - \frac{|N_G[i] \cap N_G[j]|}{|N_G[i] \cup N_G[j]|}$$

This algorithm is able to meet the criteria of proposition 1, as the maximum distance is returned when two vertices have no similarities in their closed neighborhood. It works better than Dijkstra, as it allows us to determine how similar the neighborhoods of two vectors are, and hence, how likely it is for the vertices to belong in the same cluster. In addition, the time complexity of this algorithm is significantly better than using Dijkstra's algorithm, as the worst-case time complexity is $\mathcal{O}(|N_G[V_s]|)$, with $|N_G[V_s]|$ being the size of the closed neighborhood of the vectors provided as input to the function.

This algorithm gives equal bias to all clusters, regardless of the number of edges in the cluster. However, clusters with a higher number of edges should be given priority, as making a non-optimal decision in larger cluster will lead to more cluster edits, as each vertex will need to have more edge modification in order to join the clique, leading to a larger deviation from the optimal solution.

4.1.3 Edit gain/loss

Taking into account the previously mentioned bias that we would prefer in the metric, the following equation can be used:

$$d_G(i,j) = \frac{|N_G[i] \cup N_G[j]|}{2} - |N_G[i] \cap N_G[j]|$$

The main difference of this metric would be that it prioritizes vertices that have more edges. Furthermore, it returns a negative number for vertices that require fewer edits to belong in the same closed neighborhood, and a positive for ones that require fewer edits to belong to completely different closed neighborhood. Lastly, it has the same time complexity as the closed neighborhood similarity algorithm. However, the first proposition is not guaranteed with this algorithm, as two nodes with no common vertices in their closed neighborhood are not guaranteed to have the largest distance.

4.1.4 Optimizations based on theory

In order to make the previous two metrics more accurate, we can take into account the following two propositions:

First, from Bastos et al. (2014), we have the following:

Proposition 2 *Let G be a graph, with i, j being vectors of G. If $N_G[i]$ j= $N_G[j]$, then, in an optimal solution of the cluster editing problem for G, vertex i and j are parts of the same cluster.*

And according to Komusiewicz and Uhlmann (2012):

Proposition 3 *Let G be a graph, with i, j being vectors of G. If $N_G[i] \cap N_G[j] = \{u\}$, then, in an optimal solution of the cluster editing problem for G, vertex i and j are parts of different clusters.*

Proposition 2 states that if two vertices share the same closed neighborhood, there is an optimal solution where the two vertices are in the same cluster. Proposition 3 states that if there is only a single common vertex in the closed neighborhood of two vertices, then there is an optimal solution where the two vertices are in different clusters.

We can check for these two propositions, together with proposition 1, and return the maximum possible distance, if proposition 1 or 3 were met, or the minimum possible distance, if proposition 2 was met. By guaranteeing these rules, it is more likely that the final edit will be closer to the optimal.

4.1.5 Random

Lastly, using a function that returns a random number will also be tested. This can work as a worst-case scenario, and help discover how good each of the other metrics are. The expectation is that the other algorithms will perform significantly better that this algorithm, with the exception of Dijkstra, which will only be slightly better.

4.2 Clustering algorithms

With the distance metrics decided, the other issues of the clustering algorithms discussed in section 3 can be solved. As the goal is to develop a heuristic algorithm, the clustering algorithms do not have to be exact. Preferably, the time complexity should be polynomial, and the algorithm should be able to produce an answer that is close to the best feasible result of the algorithm, even if it is not given enough time to finish.

4.2.1 K-means clustering

The first problem to be solved is deciding the number of clusters k. A graph with a high number of edges is likely to have fewer clusters, as it is more densely connected. Therefore, we can use the following formula in order to calculate the find an approximation to the actual number of clusters.

$$k_{guess} = \frac{n(n+1)}{2\,m}$$

This equation compares the total number of edges a graph can have, to the number of actual edges, in order to determine a guess for k. The original guess might not produce a good answer, however, then, the algorithm can be rerun with the next closest k to the guess.

Next is deciding how to pick the initial k-means. The fastest way to do this is randomly pick k vertices. One other way would be for the initial picked vertices to be spread out. However, this

would mean that the distance of all the vertices would have to be calculated, which would have a time complexity of $\mathcal{O}(d_G(n)^2)$, with $d_G(n)$ being the time complexity of calculating the distance. Therefore, the first solution seems more viable.

Last, deciding on the new centroids also gives us a similar dilemma. By calculating the distance of each vertex in the cluster, and using the one vertex with the lowest distance sum, this has a big toll on the time complexity of the algorithm. Picking a random vertex from each cluster as the new centroid, if using a distance metric based on the closed neighborhood, should result in the algorithm producing clusters with a low edit value.

We can add further additions to improve the algorithms. First, if a vertex follows Proposition 1 or 3 for all clusters, meaning they do not belong in any existing cluster, we can create a new cluster. This would increase k by one. Furthermore, if any of the original vertices belongs in the same cluster as another picked vertex, meaning proposition 2 was met, they will be merged, creating a single cluster.

Hence, the final code is the following:

Code 1: Algorithm for solving the cluster editing problem based on the k-means algorithm

```
k_means(G, iterations):
    best_solution
    ks = order ks, where k ∈ {Z| [1, n]}, by proximity to (n(n+1))/(2 m)
    for k in ks:
        centroids = pick k distinct random vertices from G
        clusters = map(x → [x], centroids)
        for i in iterations:
            for v in V_G:
                min_distance = (∞, None)
                for centroid in centroids:
                    dist = distance(centroid, v)
                    min_distance = min(min_distance, dist)
                    if v is centroid and min_distance[1] ≠ v:
                        merge(v.cluster, min_distance[1])
                    else if min_distance[1] is None:
                        centroids. append(v)
                        clusters. append(v)
                    else:
                        clusters[min_distance[1].cluster].append(v)
            best_solution = min(clusters, best_solution)
            centroids = update_centroids(clusters)
    return best_solution
```

The time complexity of the algorithm is $\mathcal{O}(ind_G(n))$, where i is the number of iterations, if allowed to run for all possible k. However, if the original guess for k is close the number of clusters in the best solution, the average time complexity would be $\mathcal{O}(kid_G(n))$, where k represents the value of the first few k. One major drawback of this algorithm is that the randomness of this algorithm makes this approach less reliable.

4.2.2 Agglomerative clustering

The next algorithm is based on the agglomerative hierarchical clustering algorithm. The way the clusters are merged is done by measuring the distance between the vertices of the clusters, and

combining the two clusters with the smallest distance. The algorithm will return the best cluster combination created during this process. The following pseudocode implements this algorithm:

Code 2: Algorithm for solving the cluster editing problem based on the agglomerative algorithm

```
agglomerative_clustering(G, time):
    best_solution
    clusters = clusters = map(x → [x], VG)
    while length(clusters) > 1:
        min_distance = (∞, None)
        for i in clusters:
            for j in clusters:
                dist = distance(i, j)
                min_distance = min(min_distance, dist)
        clusters = merge(min_distance[1])
        best_solution = min(clusters, best_solution)
    return best_solution
```

The worst-time complexity of this algorithm is $\mathcal{O}(n^3)$, if the merging of clusters is unbalanced. A major disadvantage of this algorithm is that because at the beginning, the rate of merging clusters is slow, as clusters contain a low number of vertices, if stopped early, the result will likely be far from the optimal solution. Another drawback of this algorithm is that when measuring distances, this occurs between all vertices of a cluster, which could slow down the performance of the algorithm.

4.2.3 Divisive clustering

The last algorithm uses the divisive approach instead. This approach selects two vertices that have the largest distance in the cluster, and splits on those two vertices, with each cluster having vertices that are closer to the original two split vertices. The algorithm can be written in pseudocode as such:

Code 3: Algorithm for solving the cluster editing problem based on the divisive algorithm

```
divisive_clustering(G, time):
    best_solution
    clusters = [[VG]]
    priority_queue = calculate all possible distances
    while length(clusters) <= n:
        left_split, right_split = priority_queue.dequeue
        if left_split not in right_split.cluster:
            continue
        cluster = right_split.cluster
        left_cluster = [left_split]
        right_cluster = [right_split]
        for i in cluster:
            add i to cluster with min distance to left_split/right_split
        clusters.remove(cluster)
        clusters.add(left_cluster, right_cluster)
        best_solution = min(clusters, best_solution)
    return best_solution
```

This algorithm has the same time complexity as the agglomerative hierarchical clustering. However, it is likely that this algorithm will be able to find an optimal solution in a faster time compared to the agglomerative algorithm, due to not having the same drawbacks.

In total the following six distance metrics will be used:

- Random
- Dijkstra's algorithm
- Closed neighborhood similarity
- Edit gain/loss
- Optimized closed neighborhood similarity
- Optimized edit gain/loss

And the following three algorithms:

- K-means
- Agglomerative hierarchical clustering
- Divisive hierarchical clustering

5 Experimental setup

Section 2 mentioned the variables of the setup. This section will mention how the test will be performed, in order to ensure that there are no biases in the results, and ensure that the controlled variables remain constant.

The following tests will be run. In the first experiment, all six distance metrics will be combined with the following algorithms: The divisive, agglomerative, and k-means algorithms with 32 iterations (i) for each k value. This produces 18 different algorithm variations, which will be run over the 100 graphs. The second experiment will go through 10 different variations of the k-means algorithm, using the optimized edit gain/loss distance metric with the following i: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512. The goal of this experiment is to discover what the best i value for the algorithm is.

The algorithms are given a deadline of 150 seconds, which should be enough time on the specific hardware for each algorithm to find the best solution they can for most graphs. The timer stars only after the function has started, and does not count reading the file from storage. The time is measured using `time.process_time()`.

Each algorithm outputs a dictionary, that represents what the best edit value was at a specific time, along with a graph, representing the graph after the edits. The time of each algorithm will be based on when what time the best answer was produced.

The quality of the final edit of each run will be determined by comparing the number of edits to the number of edits in the optimal solution. More specifically, it will be calculated how much larger as a percentage the number of edits produced is compared to the optimal.

The relevant specifications of the hardware used are the following:

- CPU: intel core i7 7700HQ @2.8GHz
- Memory: 16GB
- OS: Windows 10 Version 2004

The algorithms will be executed using pypy3.7-v7.3.5-win64, which provides a significant speedup over the default compiler/interpreter of python3, cpython (The PyPy Team, 2021).

6 Results

This section presets and discusses the results. First, the effects of each of the distance metrics are discussed. Continuing, the number of edits of each algorithm and distance metric are analyzed. Lastly, the runtime of each algorithm is investigated.

6.1 Effects of distance metrics

In order to compare the different distance metrics, the results of the first experiment are used. In particular, for each distance metric, all the runs that used that distance metric are averaged into one value.

Overall, Dijkstra and the random algorithm had similar results, while the algorithms using the closed neighborhood performed significantly better. Dijkstra on average returned an edit result 7.391 times higher from the actual solution, while random was 7.525. The closed neighborhood similarity and edit gain/loss with the optimizations mentioned in section 4 performed the best, with a mean average of 1.153 and 1.157 respectively. As proposition 1 and 2 are already met in the default closed neighborhood similarity algorithm, it performed similarly, with a mean of 1.172. Lastly, the default edit gain/loss perform worst, with a mean of 1.738, while still being significantly better than the random and Dijkstra algorithm.

From the result, we can observe that the biggest effect on performance is whether the three propositions are met. Furthermore, it seems that the edit gain/loss algorithm did not perform better than the closed neighborhood similarity algorithm, leading to the conclusion that the bias the edit gain/loss algorithm has towards prioritizing vertices with more edges did not have a major effect.

6.2 Number of edits

Next, the average number of edits from all runs are discussed. Table 1 shows the mean difference from the optimal answer for all combinations of algorithms and metrics from the first experiment, while table 2 shows the worst-case difference.

Table 1: Mean difference from optimal for each algorithm/metric combination.

Mean difference from optimal	**K-means i = 32**	**Agglomerative**	**Divisive**
Random	650.9%	653.2%	653.4%
Dijkstra	650.1%	607.8%	659.2%
Closed neighborhood similarity	6.229%	15.83%	29.71%
Edit gain/loss	164.1%	29.96%	27.34%
Optimized closed neighborhood similarity	4.076%	18.32%	23.38%
Optimized edit gain/loss	3.865%	20.04%	23.32%

Table 2: Worst-case difference from optimal for each algorithm/metric combination.

Worst-case difference from optimal	**K-means i = 32**	**Agglomerative**	**Divisive**
Random	34762.5%	34762.5%	34762.5%
Dijkstra	34762.5%	32637.5%	34743.8%
Closed neighborhood similarity	68.75%	343.2%	198.4%
Edit gain/loss	6825.0%	257.0%	300.0%
Optimized closed neighborhood similarity	68.75%	493.7%	157.1%
Optimized edit gain/loss	68.75%	492.2%	109.1%

The k-means algorithm, when combined with the edit gain/loss metric with the optimizations based on the propositions, produced the best result, with a mean of 3.865% and worst-case of 68.75%. Next, the agglomerative algorithm produced a result 15.83% higher than the exact solution, when used with the closed neighborhood similarity distance metric. More notably, the worst-case results for the agglomerative algorithm are significantly worse than the other two algorithms, as shown in table 2. This is due to the algorithm exceeding the time limit for larger graphs, before the algorithm can find a better solution. This issue is investigated in section 6.3.1 when discussing the runtime of the agglomerative algorithm.

Looking at how the number of iterations (i) over each k value affects the edit difference from figure 2, we can see that increasing the number of iterations initially has a great effect on the performance of the algorithm. However, after 32 iterations, there seems to be no improvement. As i increases after 64, the performance worsens.

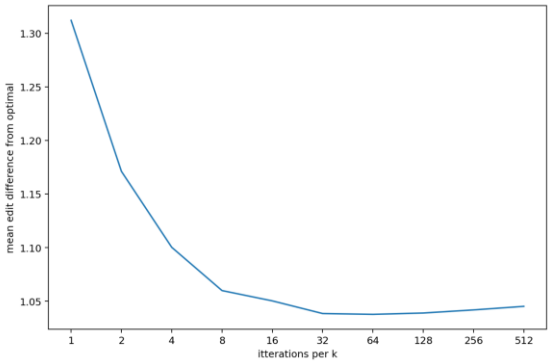

Figure 2: Mean edit difference from optimal compared to the iteration per k for the k-mean algorithm, using edit/loss optimized algorithm

For higher values, it is more likely that the algorithm will hit the deadline of 150 seconds after only checking a few k values, and before finding a good solution. Hence, lower i runs perform better on those graphs, as they iterate over more k values, and hence, find a better solution.

6.3 Runtime

Continuing, the runtime of each algorithm is investigated. Mainly observations are made on how each algorithm performs as the number of vertices increases, and how the algorithm approaches towards the best solution it can produce.

6.3.1 Agglomerative clustering

Looking at the time of the three algorithms, the agglomerative hierarchical algorithm is the slowest. As shown in figure 3, for larger problems, the agglomerative algorithm took more than 30 seconds to find the best answer it could. Overall, it seems that the complexity of the algorithm follows a cubic trend.

Except from the high complexity, other factors also played a role in the slow speed of this algorithm. The first factor is that the clusters did not have a single representative vertex, like the other two algorithms, and instead, all the vertices of a cluster were used when measuring distances. More importantly, from figure 4, it is observed that the algorithm was only able to find the optimal solution towards the end of the run. This also explains why the worst-case edit difference was so high for the agglomerative algorithm, as the algorithm did not reach the last part of the graph from figure 4 where the edit difference drops drastically, before reaching the time limit.

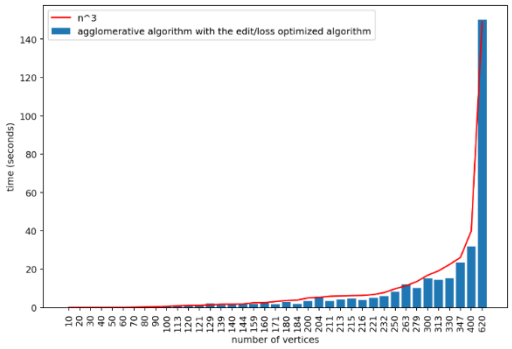

Figure 3: Number of vertices vs the time. The blue bars represent the time it took to find the best solution for the agglomerative algorithm with the edit/loss optimized distance metric. The red line shows an n^3 trend.

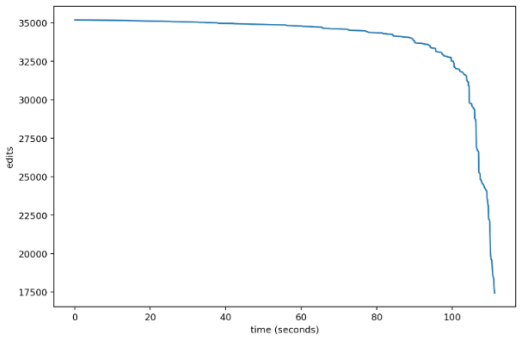

Figure 4: Time vs number of edits for a graph with $n = 620$ and $m = 35188$ with the agglomerative algorithm with the edit/loss optimized distance metric.

6.3.2 Divisive clustering

The divisive algorithm performed significantly better than the agglomerative algorithm, as seen in figure 5. Furthermore, although there is less of a pattern for this algorithm, it seems that the trend is lower than cubic. It seems that the divisive algorithm produces more balanced splits than the agglomerative algorithm, resulting in a lower average complexity. One other benefit of this algorithm, is that it approaches an optimal solution much faster, as shown in figure 6 , resulting in a faster solution.

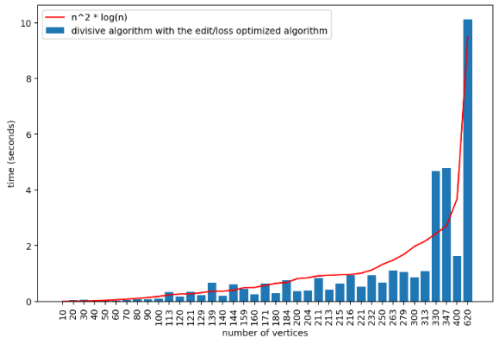

Figure 5: Number of vertices vs time. The blue bars represent time taken to find the best solution for the divisive algorithm with the edit/loss optimized distance metric. The red line shows an $n^2 * log(n)$ trend.

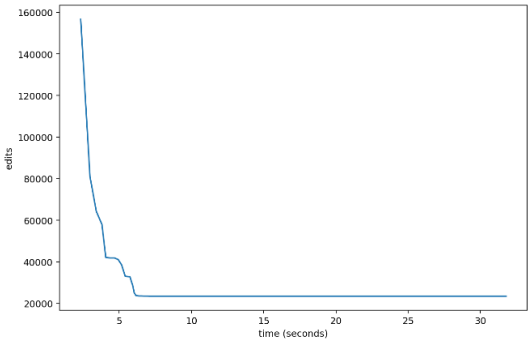

Figure 6: Time vs number of edits for a graph with n = 620 and m = 35188 with the divisive algorithm with the edit/loss optimized distance metric.

6.3.3 K-means clustering

The k-means algorithm appears to have the best performance in terms of time. This algorithm was slower than the previous two algorithms for some smaller graphs, however, it was significantly faster with larger graphs, as shown in figure 7. The slower performance on small graphs seems to be caused by the higher overhead of this algorithm. Furthermore, the randomness of this algorithm has an effect on the speed. In fact, it is much harder to discover a trend, compared to the previous two algorithms. However, the initial value of k seems to be effective, as most algorithms are able to find the best, or a close to the best answer within the first few k, as shown in figure 8. However, for some cases, the actual answer takes quite a bit longer to be found, which contributes to the lack of pattern in figure 7.

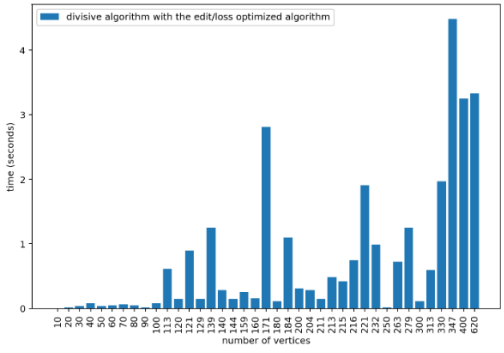

Figure 7: Number of vertices vs the time it took to find the best solution for the k-means algorithm with i = 32 and the edit/loss optimized distance metric.

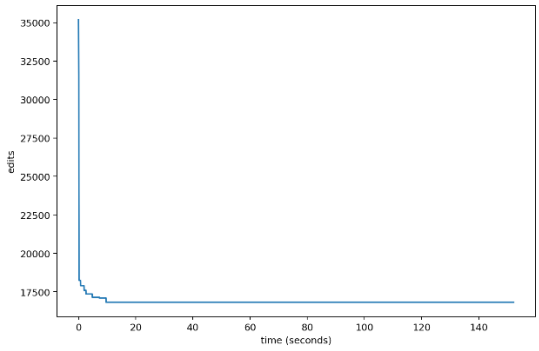

Figure 8: Time vs number of edits for a graph with n = 620 and m = 35188 with the k-means algorithm with i = 32 the edit/loss optimized distance metric.

7 Responsible research

This section discusses the ethical parts of this research. First, it is mentioned how this research can contribute to real -world applications. Next, it is mentioned how the results of this experiment can be reproduced.

The problem of heuristic cluster editing has many real-world applications, mainly in the section of biology research. Some of the implementations for cluster editing are detailed bellow:

- Help in partitioning biological data (Wittkop et al., 2010)
- Finding clusters of homologous proteins (Röttger et al., 2012)
- Network analysis of protein-protein interactions (Wittkop, Rahmann, Röttger, Böcker, & Baumbach, 2011)
- Metaolomics data analysis of ion mobility spectrometry data (Hauschild et al., 2012)

Hence, the results of this research could help push the knowledge of heuristic cluster editing algorithms, which could in turn have an impact in sections like biology.

To improve the repeatability of this research, and to allow for further research and improvements, all the code and test results are available. A GitHub page contains all the code of this research in python, as well as the results of all the tests (Zoumis, 2021).

8 Conclusion

The goal was to discover if an efficient heuristic algorithm for the problem of cluster editing could be developed, that was based on existing clustering algorithms, and to discover which of the developed algorithms had the best time complexity and proximity to result.

The main issue needed to be addressed in order to use existing algorithms for this problem was what type of distance metric should be used. From the research it was discovered that the distance metrics that were based on the closed neighborhood of the nodes we want to measure performed better than the traditional Dijkstra algorithm, which was not effective in solving this problem, as its output was similar to picking a distance randomly. The metrics that followed propositions that guaranteed whether a node would be a part of a cluster or not in an optimal solution performed the best overall.

Three algorithms were developed, that can solve the cluster editing problem at a fast time, with high accuracy. Those being the agglomerative algorithm, hierarchical algorithm, and k-means algorithm. In terms of time, although the k-means algorithm is slower for smaller graphs, the average and worst time complexity of this algorithm is lower than the other two algorithms. As a result, the k-means algorithm performed significantly better for larger graphs. As important, the k-means algorithm on average produced the result with the lowest number of edits, out of all the algorithms compared. Hence, Overall, the k-means algorithm was the most effective at solving the clustering editing problem.

8.1 Future recommendations

In the future, the k-means based algorithm could be improved. First, the algorithm could be rewritten in a lower-level language, and compared to existing heuristic algorithms, on larger graphs, in order to see how it performs then. Furthermore, the algorithm could be updated to split a graph into k large partitions. Then by treating each partition as a separate graph and using an exact algorithm to solve for the smaller partitions a solution closer to the exact answer could be found. Lastly, the k-means algorithm could be modified to generate more than one random mean for each cluster, allowing it to run in multiple threads at the same time. This would allow the algorithm to go through more iteration in the same time, improving the performance of the algorithm.

References

Bastos, L., Ochi, L. S., Protti, F., Subramanian, A., Martins, I. C., & Pinheiro, R. G. S. (2014). Efficient algorithms for cluster editing. *Journal of Combinatorial Optimization*, *31*(1), 347–371. https://doi.org/10.1007/s10878-014-9756-7

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271. https://doi.org/10.1007/bf01386390

Hamerly, G., & Elkan, C. (2002). Alternatives to the k-means algorithm that find better clusterings. *Proceedings of the Eleventh International Conference on Information and Knowledge Management - CIKM '02*. Published. https://doi.org/10.1145/584792.584890

Hauschild, A. C., Schneider, T., Pauling, J., Rupp, K., Jang, M., Baumbach, J., & Baumbach, J. (2012). Computational Methods for Metabolomic Data Analysis of Ion Mobility Spectrometry Data—Reviewing the State of the Art. *Metabolites*, *2*(4), 733–755. https://doi.org/10.3390/metabo2040733

Komusiewicz, C., & Uhlmann, J. (2012). Cluster editing with locally bounded modifications. *Discrete Applied Mathematics*, *160*(15), 2259–2270. https://doi.org/10.1016/j.dam.2012.05.019

Lloyd, S. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory*, *28*(2), 129–137. https://doi.org/10.1109/tit.1982.1056489

PACE 2021 · PACE. (2021). Retrieved April 25, 2021, from https://pacechallenge.org/2021/

Rokach, L., & Maimon, O. (2005). Clustering Methods. *Data Mining and Knowledge Discovery Handbook*, 321–352. https://doi.org/10.1007/0-387-25465-x_15

Röttger, R., Kalaghatgi, P., Sun, P., Soares, S. D. C., Azevedo, V., Wittkop, T., & Baumbach, J. (2012). Density parameter estimation for finding clusters of homologous proteins—tracing actinobacterial pathogenicity lifestyles. *Bioinformatics*, *29*(2), 215–222. https://doi.org/10.1093/bioinformatics/bts653

The PyPy Team. (2021). PyPy. Retrieved June 16, 2021, from https://www.pypy.org/

Wittkop, T., Emig, D., Lange, S., Rahmann, S., Albrecht, M., Morris, J. H., . . . Baumbach, J. (2010). Partitioning biological data with transitivity clustering. *Nature Methods*, *7*(6), 419–420. https://doi.org/10.1038/nmeth0610-419

Wittkop, T., Rahmann, S., Röttger, R., Böcker, S., & Baumbach, J. (2011). Extension and Robustness of Transitivity Clustering for Protein–Protein Interaction Network Analysis. *Internet Mathematics*, *7*(4), 255–273. https://doi.org/10.1080/15427951.2011.604559

Zoumis, A. (2021, June 23). AZoumis/Heuristic-cluster-editing. Retrieved June 27, 2021, from https://github.com/AZoumis/Heuristic-cluster-editing